OpenACC Assignment

Joseba Hernández Bravo and Jorge Vicente Puig

November 2021

Question 1

When compiling it has been identify as parallelizable the loops on lines 53, 54, 59, 60, 64 and 65. By looking at the code we can see that it refers to the main loops, i.e. laplace step, laplace error and laplace copy loops. Using `perf stat` we obtain the following measures:

- *Elapsed time:* 35,602832134 seconds.
- *Machine instructions:* 20.698.865.043.
- *IPC:* 0,17 instructions per cycle.

Is important to remark that the previous measures are done on the sequential version of the numerical method.

Question 2

When changing the `acc` option to `multicore` we are letting the compiler to parallelize the program on the CPU. Then, as before it is shown that the loops 53, 54, 59, 60, 64 and 65 are parallelizable. Moreover, we receive the following message:

```
Generating Multicore code
59, #pragma acc loop gang
```

indicating that the loop is able to run on multiple cores by using that pragma clause. Also, we recieve the message:

```
59, Generating implicit reduction(max:error)
```

showing us that a reduction clause will be used on error variable, which is necessary to recover the total error of the numerical method.

Question 3

By executing the previous compiled program on the CPU we obtain the following table,

Table 1: Measures on different executions of `laplace.c`

acc	**Elapsed time**	**Machine instructions**	**IPC**
Host	35,602832134	20.698.865.043	0,17
Multicore	109,143591715	197.810.454.874	0,16

We can see, that even though the program is parallelized we obtain worse performance, elapse time increases around 3 times. Even, the IPC is reduced on multicore program.

This decrease on performance is mainly done the way it is accessing to memory. In fact, the matrices are traversed by columns and then many time is lose on data movements from memory to the cores.

Question 4

By using the provided command we obtain the features of the NVIDIA GPU:

- *Device name:* GeForce RTX 3080.
- *Number of SMs:* 68.
- *Size of L2 cache:* 5242880 bytes.
- *Size of the device memory:* 10504306688.
- *Maximum bandwidth:* 9501 MHz * 320 bits * 2.

where the bandwidth have been obtained applying:

$$\text{bandwidth} = \text{memory clock rate} * \text{memory bus} * 2$$

Question 5

When compiling the `laplace.c` program on the NVIDIA GPU the compiler generates the code that explains the parallelization procedure.

As before, it says that loops 53, 54, 59, 60, 64 and 65 are parallelizable, also it indicates that a reduction clause have to be used to do a proper computation of the error. Moreover, we had been ask to compile the code for a GPU use and then, the memory of the Host and the GPU are different, so it generates *copy, copyin* and *copyout* messages to share data (even thought they are not necessary). Another message to look at it:

```
64, #pragma acc loop gang, vector(128) collapse(2)
/* blockIdx.x threadIdx.x collapsed-innermost */
65,   /* blockIdx.x threadIdx.x auto-collapsed */
```

In fact, on CPU parallelization it only states that the loop is parallelizable, now it first collapse the two loops as expressed on `collapse(2)` and parallelizes them together across the GPU.

Question 6

By executing the previous compiled program on the GPU we obtain the following table,

Table 2: Measures on different executions of `laplace.c`

	Elapsed time	**Machine instructions**	**IPC**
CPU: Host	35,602832134	20.698.865.043	0,17
CPU: Multicore	109,143591715	197.810.454.874	0,16
GPU	7,682890360	19.125.304.603	0,65

As shown in the table it has been done a massive parallelism which have been reduced elapse time around 5 times (from sequential program).

Question 7

Using the provided command for analyzing the statistics of the program, we obtain several tables. Looking at *CUDA Kernel Statistics* we can observed the loops that had been parallelized and ran on the GPU:

- *main_54_gpu:* Refers to laplace step loop, it takes 29% of total kernel time, exactly 19.939.139 ns.
- *main_60_gpu:* Refers to laplace error loop, it takes again 29% of kernel time, exactly 19.917.286 ns.
- *main_65_gpu:* Refers laplace copy loop, it takes 28% of kernel time, exactly 19.448.598 ns.
- *main_60_gpu_red:* Refers to the reduction operation, i.e. taking the maximum error obtained during parallelization, it takes 13% of kernel time, exactly 8.895.044 ns.

Then, the total kernel time is 68.200.067 ns.

For the memory operations we have to look at tables *CUDA Memory Operation Statistics* both by time and by size in KiB:

- *CUDA memcpy HtoD:* Refers to the data that has been copied from Host to Device (the GPU), it took 3.113.913.317 ns the 74% of memory operations time and have moved 26.195.204,000 KiB of data.
- *CUDA memcpy DtoH:* Refers to the data that has been copied from Device to Host, it took 1.056.952.841 ns the 25% of memory operations time and have moved 13.094.403,000 KiB of data.
- *CUDA memset:* This case refers to the data that has been moved using `copy()` operation, i.e. . Both execution time and data size can be worthless in compare to the total, only 78.533 ns.

Then, the total time spent on memory operations is 4.170.944.691 ns.

By looking at total spent times we observe that the time spent on memory operations is around 61 times bigger that on kernel. This tend to indicate that probably the bottleneck is found on data movements. In fact, we can confirm it by looking at the *CUDA Kernel & Memory Operations Trace* where we can follow the execution of the program and observe that is copying to Host and Device the matrices A and Anew more times than needed: one time before each loop on each iteration of the numerical method.

Question 8

By using the data directives we can update the measures table *(we refer as GPU: data to the `laplace2.c` version)*:

Table 3: Measures on different executions of `laplace.c`

	Elapsed time	**Machine instructions**	**IPC**
CPU: Host	35,602832134	20.698.865.043	0,17
CPU: Multicore	109,143591715	197.810.454.874	0,16
GPU	7,682890360	19.125.304.603	0,65
GPU: data	0,248732236	1.014.513.784	1,30

Using the provided command for analyzing the statistics of the program, we can compare the data movement with the previous version:

Table 4: Data Movements

	Elapsed time	**Time HtoD**	**Size HtoD**	**Time DtoH**	**Size DtoH**
GPU	7,682890360 s	3.114.937.977	26.195.204,000	1.056.922.820	13.094.403,000
GPU: data	0,248732236 s	7.626.801	65.536,000	5.374.387	65.536,000

With this statistics we can see that a huge improvement has been done. In fact, total time has been reduced around 30 times, data time from host to device around 408 times and from device to host around 196 times.

By looking at the trace of the program it is clearly that now is not loose the performance do to the bottleneck, the matrix is only sent at the beginning and end of the program. Moreover, during the iterations loop there are only 2 data operations of 4 bytes which refers to a float, i.e. the variable error that is used on reduction clause.

Question 9

Execute the previous program, but increasing the number of iterations of the convergence loop from 100 to 10,000. Explain the performance results.

Table 5: Measures on different executions of `laplace.c`

Nº iterations	**Elapsed time**	**IPC**
100	0,248732236	1,30
1000	0,930517133	1,86
2500	1,9709825443	1,94
5000	3,718061370	1,92
7500	5,520748803	1,95
10000	7,268397317	2,00

As shown in previous table firstly, we can see how big the improvement using the data directives has been made, it takes same execution time for the normal version with 100 iterations than for the optimized version with 10.000 iterations.

Secondly, by looking at the IPC we are able to compare the executions between different problems sizes and observe that performance gets better when increasing the problem size. Which make sense, in fact GPUs are focus on massive parallelization instead of fast response time, so when increasing the problem size we can make the most of its potential.

Optimizing codes

We will start discussing the optimization using OpenACC of the `lap.c` code.

The first step consist of trying to parallelize the functions *laplace step* and *laplace copy* that follow the same structure: two independent loops that make some function on one element of the matrix. For that purpose, we will start by using the pragma clause `#pragma acc kernels`, which will tell us that the loop is not parallelizable, even thought we know it is.

Then, we will need to specify that loops are independent, by `#pragma acc loop independent` on both loops which will make the compiler understand the parallelization process and will by substituted by:

```
15, #pragma acc loop gang, vector(128) collapse(2)
/* blockIdx.x threadIdx.x collapsed-innermost */
16,   /* blockIdx.x threadIdx.x collapsed */
```

However, we are moving data using pointers, and then the compiler is not able to know if the directions that receive correspond to the data. For that problem, we have to use a `present(in,out)` clause, which will tell the compiler that the data is already present on device in another region.

However, this changes won't work without a proper data exchange between the GPU and the Host, which can be obtained using the clause `#pragma acc data copy(A) create(Anew)` before the while loop.

The other function that we have to parallelize is the *laplace error* function. It follows the same idea as before but we also need to use a `#pragma acc reduction(max:error)` to take care of computing a correct error. Nevertheless, the compiler will take care of it using firstly `Generating implicit copy(error)` and then `Generating implicit reduction(max:error)`.

The improvements can be seen on next table,

Table 6: Measures with changes done on `lap.c`

Changes	**Elapsed time**	**IPC**
Original version	119,547587266	0,10
Change laplace step and copy	23,797547000	0,21
All changes	0,276376348	1,34

We can see that it has been done a massive improvement on this program, the time has been reduced around 432 times from the sequential version and the IPC has increased 13,4 times. We can even use the `profGPU2.txt` and check that there is not a bottleneck on data exchange.

Table 7: Different number of iterations on `lap.c`

Nº iterations	**Elapsed time**	**IPC**
100	0,276376348	1,34
1000	0,906421167	1,82
2500	1,993668528	1,93
5000	3,787090755	1,96
7500	5,599828950	1,92
10000	7,403937936	1,96

As we can see in the above table, results are very similar with the ones obtained on exercise 9. When increasing the problem size, looking at the IPC, we can see that the program gets better results, as expected for the GPU use (small data lots of iterations).

Now, we can discuss the optimization on `lap2.c` code.

For optimizing this code the main focus will be on *laplace step* function, which both computes the error and do the step. The parallelization, similar as in previous case, will be obtained by combining a `#pragma acc kernels` with

- `loop independent collapse(2)`: As before the compiler is not sure to be independent the loops.
- `present(old[:n*m],in[:n*m])`: for telling the compiler that the data is already in another data region, including the size of the data.

the main difference is because we have to use the command `#pragma acc data copy(error)` for being able to parallelize the computation of the error.

We also include a command `#pragma acc data copy(A[:n*m]) copyin(Anew[:n*m])` to avoid unnecessary data movements.

With this optimizations we have obtained the following results,

Table 8: Different number of iterations on `lap2.c`

Nº iterations	**Elapsed time**	**IPC**
100	0,234205481	1,08
1000	0,534389401	1,60
2500	1,024441582	1,75
5000	1,869333484	1,85
7500	2,688412531	1,88
10000	3,520379494	1,93

We can see that for a low number of iterations there is not a significance performance change in comparison to `lap.c`. However, when we increase the number of iterations we can see that the performance gets better, going from around 1,17 times to 2,1 times on 10.000 iterations.